AER1418: Assignment 4

Geoff Donoghue April 26, 2019

Part 1. Advection-diffusion equation

(a) We wish to show that $\exists \gamma < \infty$ such that

$$a_h(w,v) \equiv a(w,v) + (\tau \mathcal{L}w, \mathcal{L}v)_{L^2(\Omega)} \leq \gamma \|w\|_{\mathcal{V}_h} \|v\|_{\mathcal{V}_h}, \quad \forall w, v \in \mathcal{V}_h.$$

Since we already know that

$$a(w,v) \leq \left(\|\kappa\|_{L^\infty(\Omega)} + \|b\|_{L^\infty(\Omega)} + C_{\text{tr}}^2 \|b\|_{L^\infty(\Gamma_N)} \right) \|w\|_{\mathcal{V}_h} \|v\|_{\mathcal{V}_h},$$

we seek some $\gamma' < \infty$ such that

$$(\tau \mathcal{L}w, \mathcal{L}v)_{L^2(\Omega)} \leq \gamma' \|w\|_{\mathcal{V}_h} \|v\|_{\mathcal{V}_h}, \quad \forall w, v \in \mathcal{V}_h.$$

We first apply the Cauchy-Schwarz inequality to the leas-squares term, obtaining

$$(\tau \mathcal{L}w, \mathcal{L}v)_{L^2(\Omega)} \leq \|\tau \mathcal{L}w\|_{L^2(\Omega)} \|\mathcal{L}v\|_{L^2(\Omega)}.$$

Now using the definition of $\mathcal{L}$, we can say that

$$(\tau \mathcal{L}w, \mathcal{L}v)_{L^2(\Omega)} \leq \|\tau(-\nabla \cdot (\kappa \nabla w) + \nabla \cdot (bw))\|_{L^2(\Omega)} \| - \nabla \cdot (\kappa \nabla v) + \nabla \cdot (bv)\|_{L^2(\Omega)}.$$

By applying the triangle inequality to the norms containing sums we arrive at

$$(\tau \mathcal{L}w, \mathcal{L}v)_{L^2(\Omega)} \leq (\| - \tau \nabla \cdot (\kappa \nabla w)\|_{L^2(\Omega)} + \|\tau \nabla \cdot (bw)\|_{L^2(\Omega)})(\| - \nabla \cdot (\kappa \nabla v)\|_{L^2(\Omega)} + \|\nabla \cdot (bv)\|_{L^2(\Omega)}).$$

If we assume that our advection and diffusion fields are constant we may reexpress the above as

$$(\tau \mathcal{L}w, \mathcal{L}v)_{L^2(\Omega)} \leq \tau(\kappa \|\nabla^2 w\|_{L^2(\Omega)} + b \cdot \|\nabla w\|_{L^2(\Omega)})(\kappa \|\nabla^2 v\|_{L^2(\Omega)} + b \cdot \|\nabla v\|_{L^2(\Omega)}).$$

Using the definitions of the $H^2(\Omega)$ and $H^1(\Omega)$ semi-norms we can say that

$$(\tau \mathcal{L}w, \mathcal{L}v)_{L^2(\Omega)} \leq \tau(\kappa |w|_{H^2(\Omega)} + b \cdot |w|_{H^1(\Omega)})(\kappa |v|_{H^2(\Omega)} + b \cdot |v|_{H^1(\Omega)}).$$

Since we have $\mathcal{V}_h \subset H^1(\Omega)$, $\overline{\Omega} = \bigcup_{K \in \mathcal{T}_h} \overline{K}$, and we assume the inverse esimtate estmate $|v|_{H^2(K)} \leq c_{\text{inv}} h^{-1} \|v\|_{H^1(K)} \forall v \in \mathcal{V}_h$ our inequality now becomes

$$(\tau \mathcal{L}w, \mathcal{L}v)_{L^2(\Omega)} \leq \tau \Sigma_{K \in \mathcal{T}_h} (\kappa c_{\text{inv}} h^{-1} \|w\|_{H^1(K)} + b \cdot |w|_{H^1(\Omega)})(\kappa c_{\text{inv}} h^{-1} \|v\|_{H^1(K)} + b \cdot |v|_{H^1(\Omega)}).$$

From the definition of the $H^1(\Omega)$ seminorm, $\|v\|_{H^1(\Omega)} \equiv \|v\|_{L^2(\Omega)} + |v|_{H^1(\Omega)}$, $\forall v \in H^1(\Omega)$, we note that by the non-negativity of $\|v\|_{L^2(\Omega)}$ implies that $\|v\|_{H^1(\Omega)} \geq |v|_{H^1(\Omega)}$, applying this to our inequality we arrive at

$$(\tau \mathcal{L}w, \mathcal{L}v)_{L^2(\Omega)} \leq \tau \Sigma_{K \in \mathcal{T}_h} (\kappa c_{\text{inv}} h^{-1} \|w\|_{H^1(K)} + b \cdot \|w\|_{H^1(\Omega)})(\kappa c_{\text{inv}} h^{-1} \|v\|_{H^1(K)} + b \cdot \|v\|_{H^1(\Omega)}).$$

$$(\tau \mathcal{L}w, \mathcal{L}v)_{L^2(\Omega)} \leq \tau \Sigma_{K \in \mathcal{T}_h} (\kappa c_{\text{inv}} h^{-1} + b) \|w\|_{H^1(K)} (\kappa c_{\text{inv}} h^{-1} + b) \|v\|_{H^1(K)}.$$

Since when we invoked the inverse estimate inequality, we can now express τ in the limit as $h \to 0$, which is $\tau = \dfrac{h^2}{12\kappa}$, this updates our inequality to become

$$(\tau \mathcal{L}w, \mathcal{L}v)_{L^2(\Omega)} \leq \frac{h^2}{12\kappa} (\kappa c_{\text{inv}} h^{-1} + b)^2 \Sigma_{K \in \mathcal{T}_h} \|w\|_{H^1(K)} \|v\|_{H^1(K)}.$$

Since $h \to 0$ the only term that survives is

$$(\tau \mathcal{L}w, \mathcal{L}v)_{L^2(\Omega)} \leq \frac{\kappa}{12} c_{\text{inv}}^2 \|w\|_{H^1(\Omega)} \|v\|_{H^1(\Omega)}.$$

Therefore we say that the bilinear form of the advection-diffusion equation arising from the GLS discretization is continuous with a continuity constant

$$\gamma = \|\kappa\|_{L^\infty(\Omega)} + \|b\|_{L^\infty(\Omega)} + C_{\text{tr}}^2 \|b\|_{L^\infty(\Gamma_N)} + \frac{\kappa}{12} c_{\text{inv}}^2.$$

(b) Assuming a constant diffusion field, our problem is given by

$$-\kappa\left(\frac{\partial^2 u}{\partial x_1^2}+\frac{\partial^2 u}{\partial x_2^2}\right)+\frac{\partial u}{\partial x_1}=0.$$

We note that this problem is separable and express our solution as the product of monomial functions $u = X_1X_2$ and obtain

$$-\kappa\frac{1}{X_1}\frac{d^2X_1}{dx_1^2}+\frac{1}{X_1}\frac{dX_1}{dx_1}=\kappa\frac{1}{X_2}\frac{d^2X_2}{dx_2^2}=m,$$

where m is some seperation constant. The ordinary differential equation in x_2,

$$\frac{d^2X_2}{dx_2^2}=\frac{m}{\kappa}X_2,$$

is trivial to solve; the solution is given by

$$X_2(x_2)=c_1\exp\left(\sqrt{\frac{m}{\kappa}}x_2\right)+c_2\exp\left(-\sqrt{\frac{m}{\kappa}}x_2\right).$$

Our boundary conditions state that

$$\left.\frac{dX_2}{dx_2}\right|_{x_2=0}=0=c_1\sqrt{\frac{m}{\kappa}}-c_2\sqrt{\frac{m}{\kappa}}\implies c_1=c_2,$$

$$\left.\frac{dX_2}{dx_2}\right|_{x_2=1}=0=c_1\sqrt{\frac{m}{\kappa}}\exp\left(\sqrt{\frac{m}{\kappa}}\right)-c_1\sqrt{\frac{m}{\kappa}}\exp\left(-\sqrt{\frac{m}{\kappa}}\right)\implies c_1\vee m=0.$$

Since either c_1 or m will result in the trivial solution, we will take $m = 0$ and see if the resulting solution for u satisfies our differential equation, then invoke Lax-Milgram to justify the uniqueness of the solution. By taking m to be zero, our ordinary differential equation in x_2 is updated to become

$$\frac{d^2X_2}{dx^2}=0,\quad \left.\frac{dX_2}{dx_2}\right|_{x_2=0}=0,\quad \left.\frac{dX_2}{dx_2}\right|_{x_2=1}=0.$$

Here we note that any constant function will satisfy all of these equations, the magnitude of this constant can be absorbed into X_1 and we will therefore say that $X_2 = 1$.

Our ordinary differential equation in x_1 is then given by

$$-\kappa\frac{d^2X_1}{dx_1^2}+\frac{dX_1}{dx_1}=0,\quad X_1(x_1=0)=1,\quad X_1(x_1=1)=0,$$

which has the characteristic equation

$$-\kappa r^2+r=0,\implies r=\frac{1}{\kappa}.$$

Assuming a solution given by a linearly independent combination of exponentials, we can apply our boundary conditions to obtain

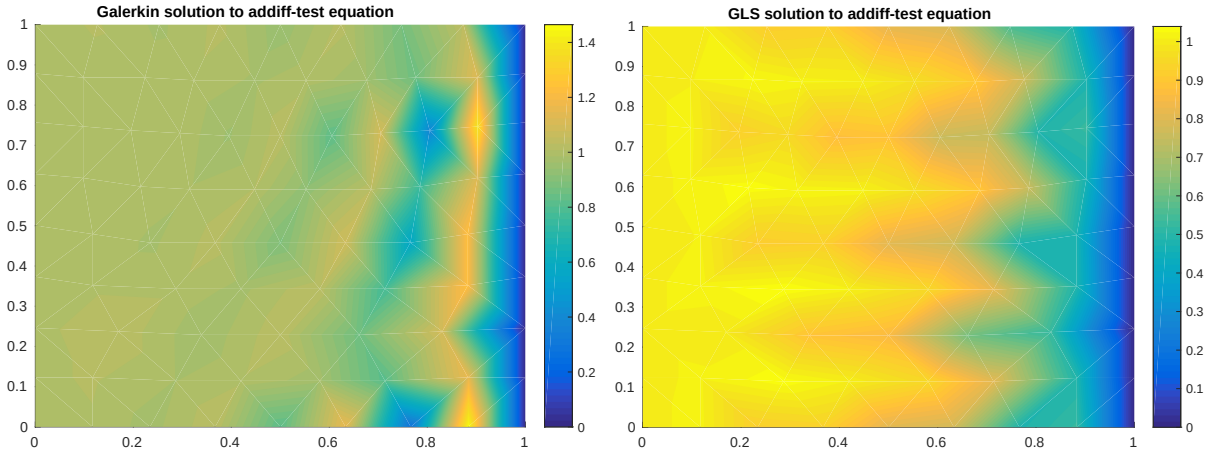
$$X_1(x_1)=\frac{e^{x_1/\kappa}-e^{1/\kappa}}{1-e^{1/\kappa}}.$$

The closed form solution of our advection-diffusion equation is

$$u(x_1,x_2;\kappa)=X_1(x_1;\kappa)X_2(x_2)=\frac{e^{x_1/\kappa}-e^{1/\kappa}}{1-e^{1/\kappa}}.$$

(c-f) See appendix.

(g) The standard continuous Galerkin and Galerkin least-squares solution fields for the above equation using a $\mathbb{P}^2$ approximation are shown in Figue 1.



(a) Without GLS Stabilization (b) With GLS Stabilization

Figure 1: Galerkin approximations to the advection-diffusion test equation.

We note qualitatively for now that the unstabilized method has a maximum overshoot that is greatly reduced by applying Galerkin least-squares stabilization.
The convergence of the $L^2(\Omega)$ norm of the solution error is shown in Figure 2 below.

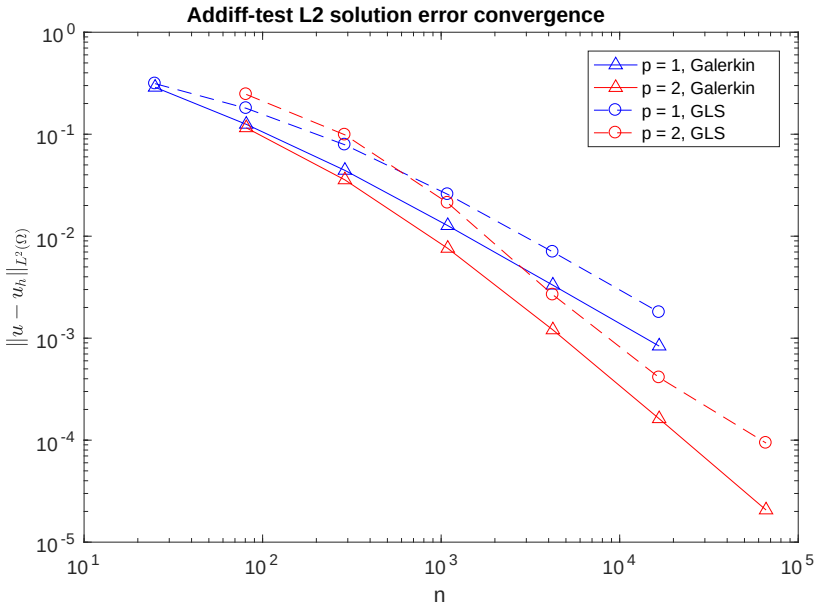


Figure 2: $L^2(\Omega)$ error convergence, $\mathbb{P}^1$, and $\mathbb{P}^2$

The asymptotic convergence rates for each scheme are summarized in table 1. We note that the expected convergence rates with respect to the n for the $L^2(\Omega)$ norm of the error are $h^{p+1} \sim n^{-(p+1)/2}$. Moreover, for the GLS scheme we expect a supoptimal convergence for the $L^2(\Omega)$ norm of the error $\sim n^{-(p+1/2)/2}$ but we may hope to recover the optimal convergence rate for this problem if it is sufficiently smooth.

	$\mathbb{P}^1$	$\mathbb{P}^2$
Expected Galerkin	-1.00	-1.50
Expected GLS	-1.00	-1.50
Actual Galerkin	-0.9975	-1.4789
Actual GLS	-0.9768	-1.3638

Table 1: Convergence rates for each of the schemes.

Here we note that all of the convergence rates are in good agreement with the expected finite element theory, and that the true solution is apparently smooth enough to yield the optimal convergence rate for GLS.
Lastly, we consider the maximum overshoot in the FE coefficients, these results are shown in Figure 3.

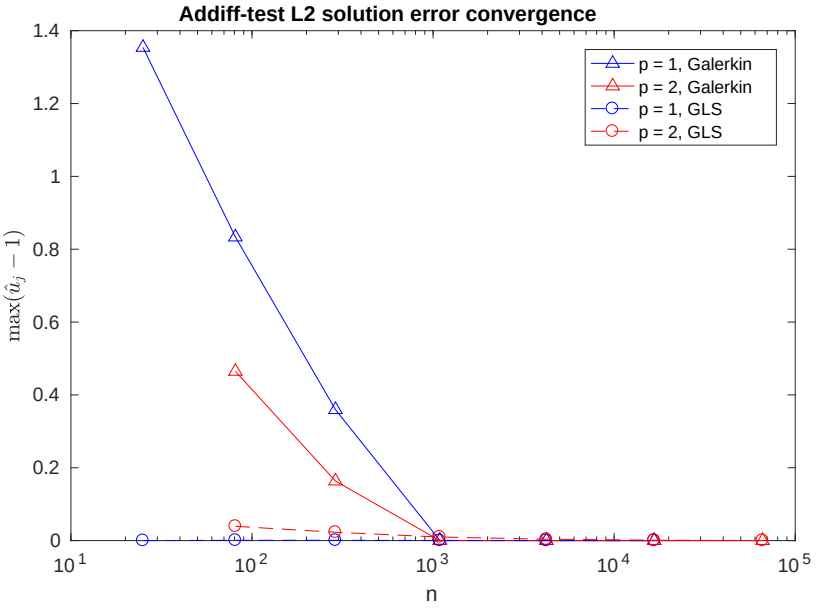


Figure 3: Maximum overshoot, for standard Galerkin and GLS with $\mathbb{P}^1$ and $\mathbb{P}^2$

It is clear that the stability term introduced by GLS greatly reduces the maximum overshoot for coarse meshes. This difference becomes minimized as the mesh is refined.

Part 2. Error estimation and adaptive mesh refinement

(a) See appendix.

(b) Adaptive convergence plots for the test advection-diffusion problem are shown in Figure 4.

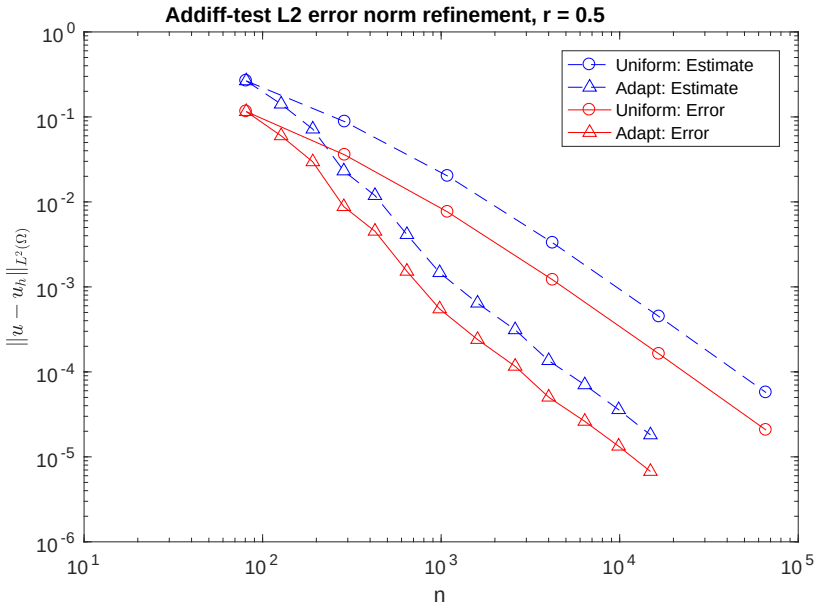


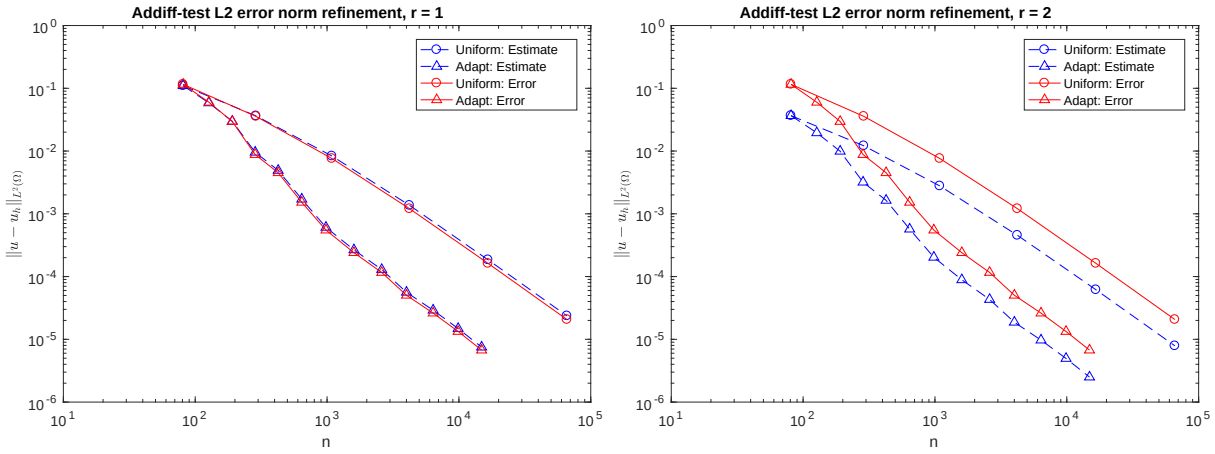
Figure 4: This is a caption. r = 0.5

The convergence rates are showin in Table 2.

	Uniform	Adaptive
Expected	-1.50	-1.50
Actual	-1.4925	-1.6017

Table 2: Convergence rates for the uniform and adaptive refinement algorithms.

The rate parameter used for the error estimates was $r = 1/2$, in order to justify this we can consider the error estimates using other rate parameters $r = 1, r = 2$. These results are shown in Figure 5 below.



(a) $r = 1$ (b) $r = 2$

Figure 5: Convergence of Galerkin and GLS schemes for $r = 1$, and $r = 2$.

We see that for $r = 2$ our error estimates are less than the actual error, while $r = 1$ provides very good estimates. We will remain using $r = 1/2$ in order to be as robust as possible.

(c) See appendix.

(d) The reference value for this problem was found to be 0.2984344669 with an associated error of $9.3389521479 \times 10^{-11}$; this was determined by running our adaptive output-based algorithm for 17 iterations. The $L^2(\Omega)$- and output-adapted meshes, along with the primal solution, are shown in Figures 6 and 7 below, respectively.

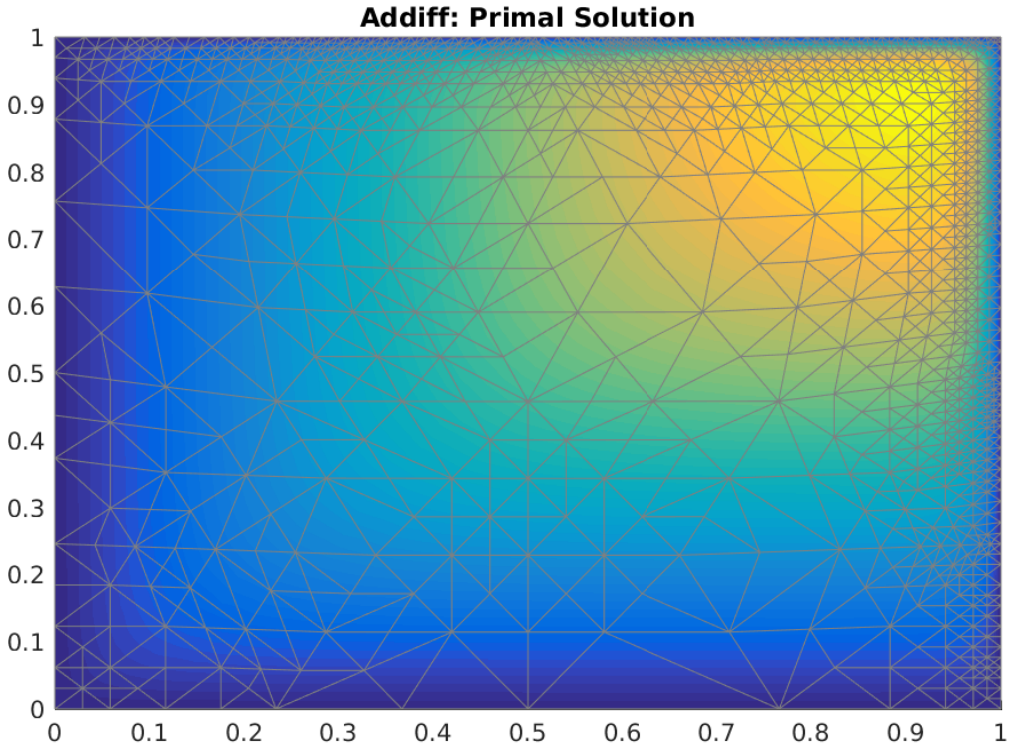


Figure 6: Primal addiff solution with mesh, L2 Adapted

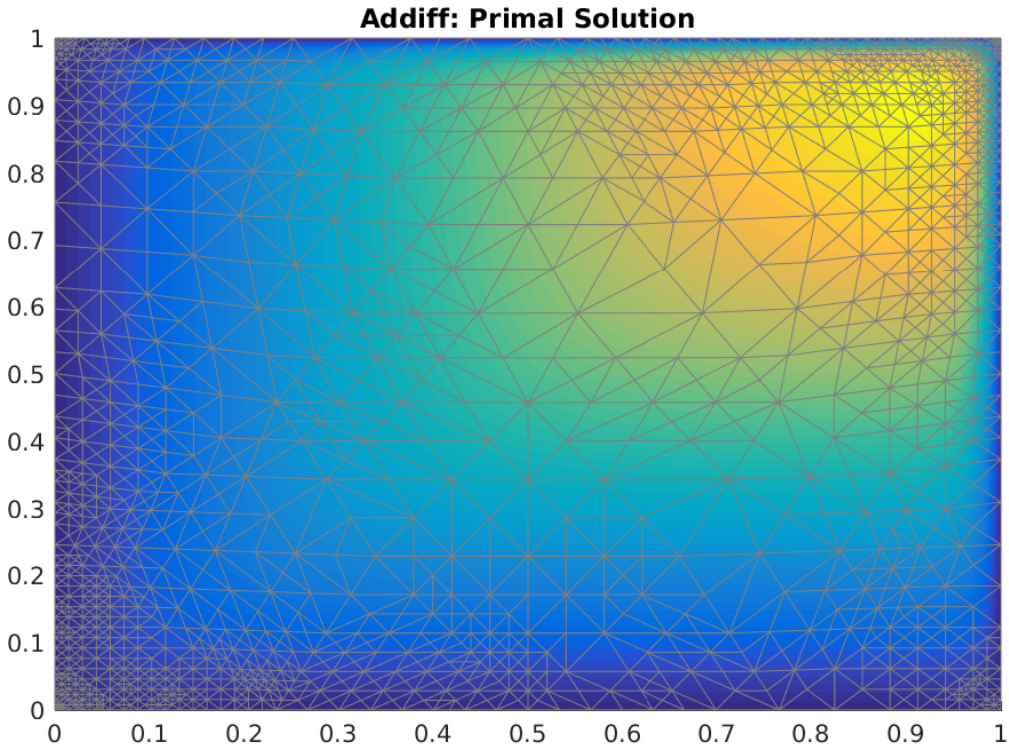


Figure 7: Primal addiff solution with mesh, Output Adapted

We note that the adaptation driven by the error in the L^2 norm of the solution adapts mostly in the top and right areas of the mesh. This can be expected as this is where the solution gradient is greatest. In the bottom and left regions the solution can be more accurately captured by a cheaper approximation.
This can be constrasted with the output-based scheme; here it helps to look at the adjoint solution of the problem, as shown in Figure 8 below.

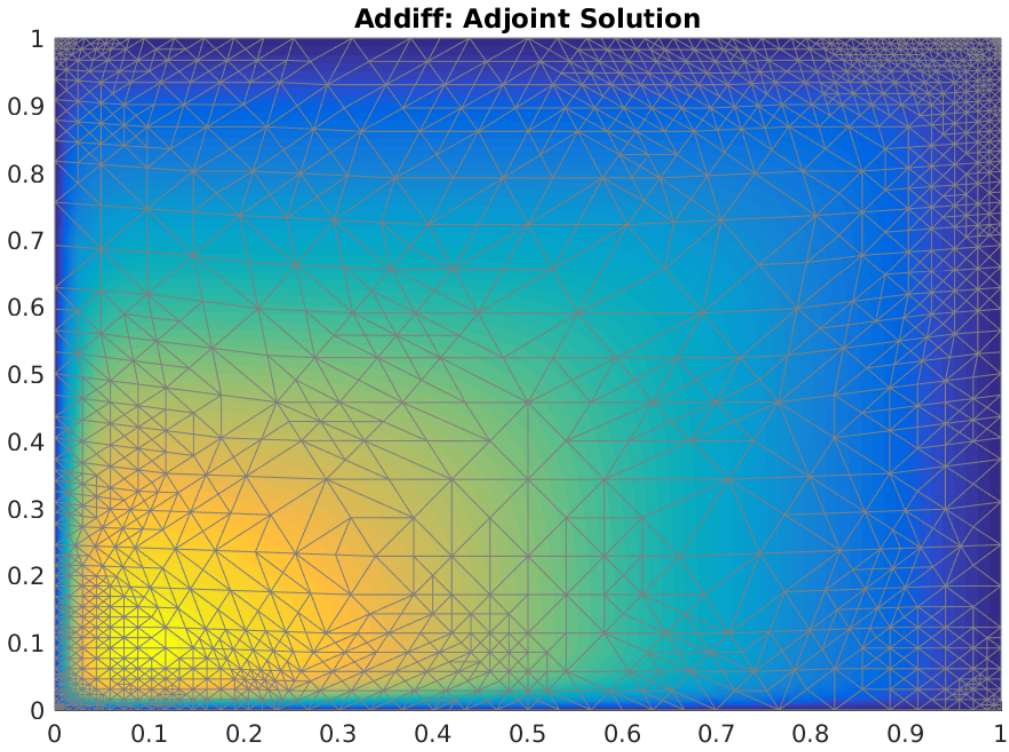


Figure 8: Adjoint solution with mesh, Output Adapted

Here we note that when we weigh our indicators with the adjoint and primal solutions we obtain a more evenly distributed refinement strategy, with a slight preference to the edges of the domain. Again, we see that the gradient of the adjoint is greatest in the bottom left corner and the combination of this along with the primal error discussed previously is what yields the combined refinement strategy.

Lastly, we plot the output error as a function of n for uniform, L^2-adaptive, and output-adaptive refinement in Figure 9 below.

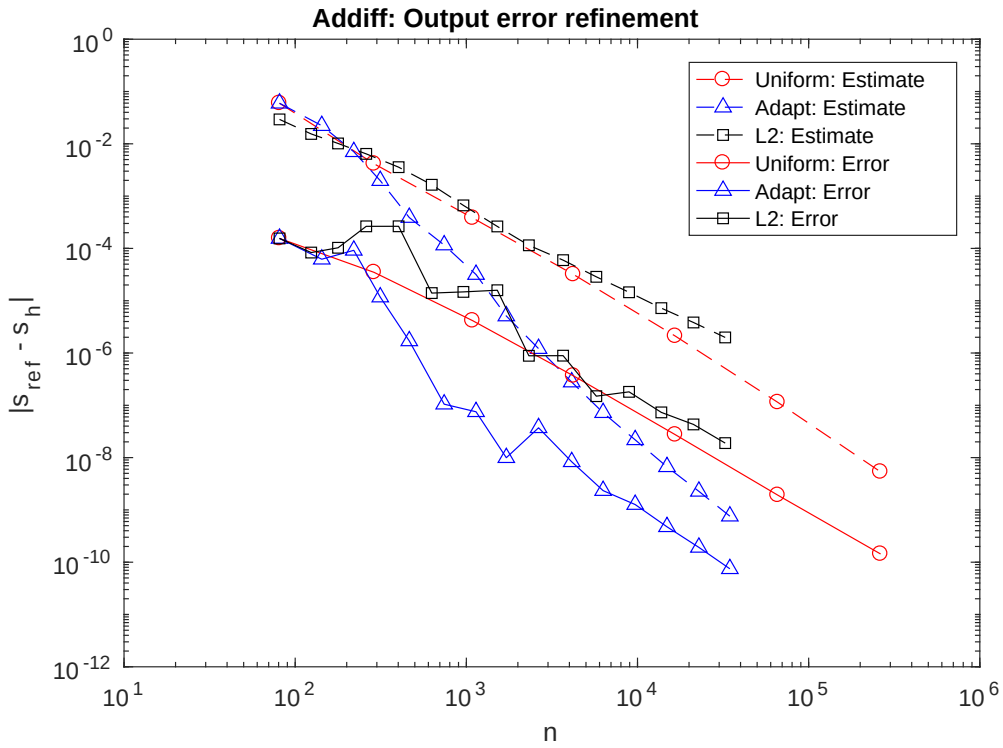


Figure 9: Addiff refinemetn

The convergence rates are summarized in Table 3 below.

	Uniform	L^2	output
Expected	-2.00	-2.00	-2.00
Actual	-2.2125	-1.542	-2.5700

Table 3: Convergence rates for each of the adaptation algorithms

Here we see that the expected convergence rate of $2p$ is in moderate agreement with the uniform and output-adaptive algorithms while the L^2-scheme falls slightly short. We can speculate that this is a result of the fact that the output error depends on both the error in the adjoint, and the L^2 strategy will choose to refine only the error in the primal solution. This will cause the error in the adjoint to dominate while additional refinement is spent in areas of the domain that do not require it.

Part 3. Adaptive eigensolver

(a) The weak form of our eigenproblem on $\Omega \equiv [0,1]^2$ can be stated as: find $(u_k, \lambda_k) \in H_0^1(\Omega) \times \mathbb{R}, k \in \mathbb{Z}_{>0}$ such that $\|u_k\|_{L^2(\Omega)}$ and

$$\int_\Omega \nabla v \cdot \nabla u_k dx = \lambda_k \int_\Omega v u_k dx \quad \forall v \in H_0^1(\Omega).$$

Integrating by parts the left side of the weak form of our eigenproblem above, we arrive at

$$\int_{\partial\Omega} v(n \cdot \nabla u) ds - \int_\Omega v \nabla^2 u_k dx = \lambda_k \int_\Omega v u_k dx \quad \forall v \in H_0^1(\Omega),$$

a simple rearrangement yields

$$\int_{\partial\Omega} v(n \cdot \nabla u) ds - \int_\Omega v(\nabla^2 u_k + \lambda_k u_k) dx = 0 \quad \forall v \in H_0^1(\Omega).$$

The strong form of the above is given by

$$-\nabla^2 u_k = \lambda_k u_k \text{ in } \Omega, \quad n \cdot \nabla u_k = 0 \text{ on } \partial\Omega.$$

This PDE is not separable but since it is in a form amenable to periodic solutions, let us guess the solution $u_k = A_0 \sin(k\pi x_1) \sin(k\pi x_2)$ — A_0 being some normalization constant — and test our hypothesis. We note that

$$\nabla^2 u_k = 2k^2\pi^2 u_k, \text{ and } n \cdot \nabla u_k = 0, \forall k \in \mathbb{Z}.$$

We note that since we require $\|u_k\|_{L^2(\Omega)} = 1$, and

$$\|u_k\|_{L^2(\Omega)} = \int_0^1 \int_0^1 (\sin(k\pi x_1) \sin(k\pi x_2))^2 dx_1 dx_2 = \frac{1}{4} \implies A_0 = 2,$$

our closed form expression for our first eigenpair on our unit-square domain is

$$(u_1, \lambda_1) = (2 \sin(\pi x_1) \sin(\pi x_2), 2\pi^2).$$

(b) Let us define the eigenproblem on the arbitrary domain $\tilde{\Omega}$ to be

$$\int_{\tilde{\Omega}} \tilde{\nabla} v \cdot \tilde{\nabla} u_k d\tilde{x} = \tilde{\lambda_k} \int_{\tilde{\Omega}} v u_k d\tilde{x}.$$

We note that from the definition of our scaled domain Ω we have

$$dx = H d\tilde{x}, \implies \tilde{\nabla} = \frac{\partial}{\partial \tilde{x}} = \frac{\partial}{\partial x}\frac{\partial x}{\partial \tilde{x}} = H\nabla$$

This allows us to take our eingenproblem to be

$$\int_\Omega H\nabla v \cdot H\nabla u_k \frac{dx}{H} = \tilde{\lambda_k} \int_\Omega v u_k \frac{dx}{H}.$$

$$\int_\Omega \nabla v \cdot \nabla u_k dx = \underbrace{\frac{\tilde{\lambda_k}}{H^2}}_{\lambda_k} \int_\Omega v u_k dx.$$

Here we notice that $\lambda_k = \dfrac{\tilde{\lambda_k}}{H^2}$, and $q = -2$. As a corolloray, we can note that $\lambda_{\min}^{\mathrm{S}} = \dfrac{\lambda_1}{H^2} = \dfrac{\pi^2}{2}$

(c) We note that the smallest eigenvalue can be taken as a coercivity constant, i.e. the coercivity constant can be interpreted as the lower bound of the Rayleigh quotient, the lower bound of which is given by the smallest eigen value, expressed mathematically this says

$$\lambda_{\min} = \inf_k \{\lambda_k\} = \alpha = \inf_{v \in \mathcal{V}_h} \frac{a(v,v)}{\|v\|^2_{\mathcal{V}_h}}.$$

Applying this to our L-shaped and square domains — Ω^{L} and Ω^{S}, respectively — we obtain the following expressions

$$\lambda^{\mathrm{S}}_{\min} = \inf_{v \in H^1_0(\Omega^{\mathrm{S}})} \frac{a^{\mathrm{S}}(v,v)}{\|v\|^2_{H^1_0(\Omega^{\mathrm{S}})}}, \quad \lambda^{\mathrm{L}}_{\min} = \inf_{v \in H^1_0(\Omega^{\mathrm{L}})} \frac{a^{\mathrm{L}}(v,v)}{\|v\|^2_{H^1_0(\Omega^{\mathrm{L}})}}.$$

Where $a^{\mathrm{L}}(v,v), a^{\mathrm{S}}(v,v)$ are the bilinear form for the Rayleigh quatients on each space. We note that since we have $H^1_0(\Omega^{\mathrm{L}}) \subset H^1_0(\Omega^{\mathrm{S}})$ we can say that

$$\lambda^{\mathrm{S}}_{\min} = \inf_{v \in H^1_0(\Omega^{\mathrm{S}})} \frac{a^{\mathrm{S}}(v,v)}{\|v\|^2_{H^1_0(\Omega^{\mathrm{S}})}} \leq \inf_{v \in H^1_0(\Omega^{\mathrm{L}})} \frac{a^{\mathrm{L}}(v,v)}{\|v\|^2_{H^1_0(\Omega^{\mathrm{L}})}} = \lambda^{\mathrm{L}}_{\min}.$$

This demonstrates that $\lambda^{\mathrm{S}}_{\min} \leq \lambda^{\mathrm{L}}_{\min}$.

(d) We define $\Omega^{\mathrm{U}} \equiv [0,1]^2$ to be the unit square and note that our square domain Ω^{S} can be expressed as

$$\Omega^{\mathrm{S}} = \{x \in \mathbb{R}^2 : x = H\tilde{x} = 2\tilde{x}, \tilde{x} \in \Omega^{\mathrm{U}}\}.$$

We note from part (a) that $\lambda^{U}_{\min} = \dfrac{\pi^2}{2}$, and that

$$\lambda^{S}_{\min} = H^{-2}\lambda^{U}_{\min} = \frac{\pi^2}{2}.$$

Finally from part (c) we can say that

$$\lambda^{\mathrm{S}}_{\min} \leq \lambda^{\mathrm{L}}_{\min} \implies \frac{\pi^2}{2} \leq \lambda^{\mathrm{L}}_{\min}.$$

Which is our upper bound for the smallest eigenvalue on the L-shaped domain.

(e-f) See appendix.

(g) A convergence plot for the L^2 norm of the error in the first eigenfunction on the square domain is shown in Figure 10.

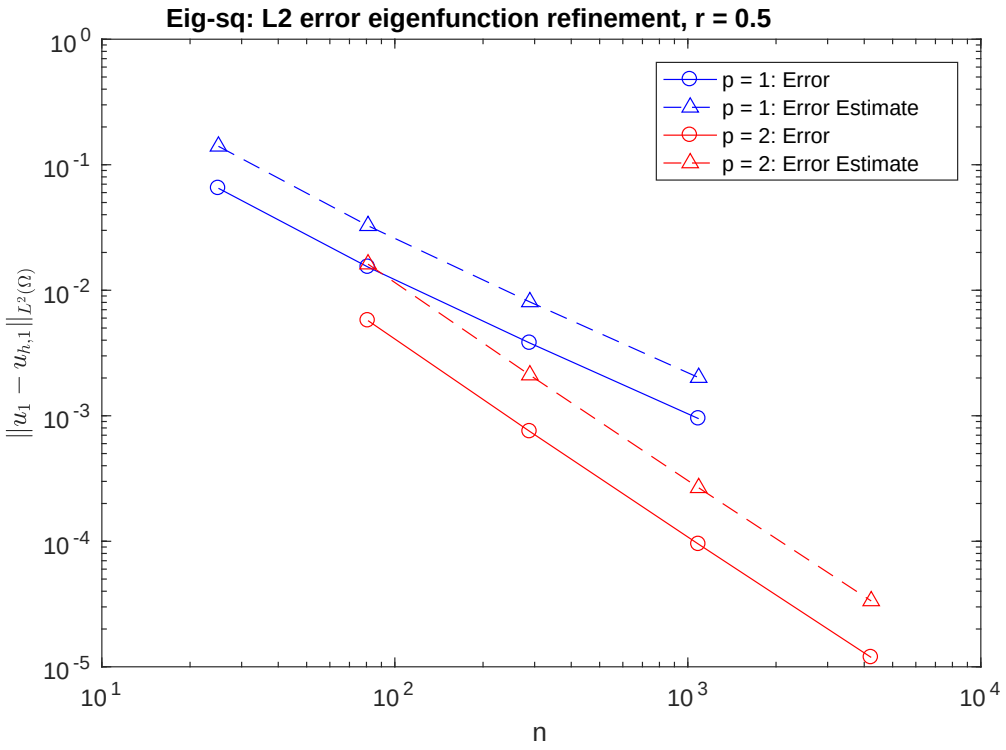


Figure 10: Square domain first eigenfunction L^2 norm convergence

Convergence rates for the L^2 norm of the error of the first eigenfunction are shown below in Table 4, we note that they agree very well with the expected value of $(p+1)/2$.

	$\mathbb{P}^1$	$\mathbb{P}^2$
Expected	-1.00	-1.50
Actual	-1.0253	-1.5332

Table 4: Convergence rates for eigenfunction error on square domain

When using a rate parameter of $r = 1/2$ the effectivities for the $\mathbb{P}^1$ and $\mathbb{P}^2$ schemes are ~ 2.13 and ~ 2.82 respectively, both happen to be quite constant. Switching to $r = 1$, the effectivities are still just as constant, but they are updated to become ~ 0.89 and ~ 1.16 for $\mathbb{P}^1$ and $\mathbb{P}^2$ respectively.

With regard to eigenvalues, their error as a function of n for uniform refinement are shown in Figure 11 below.

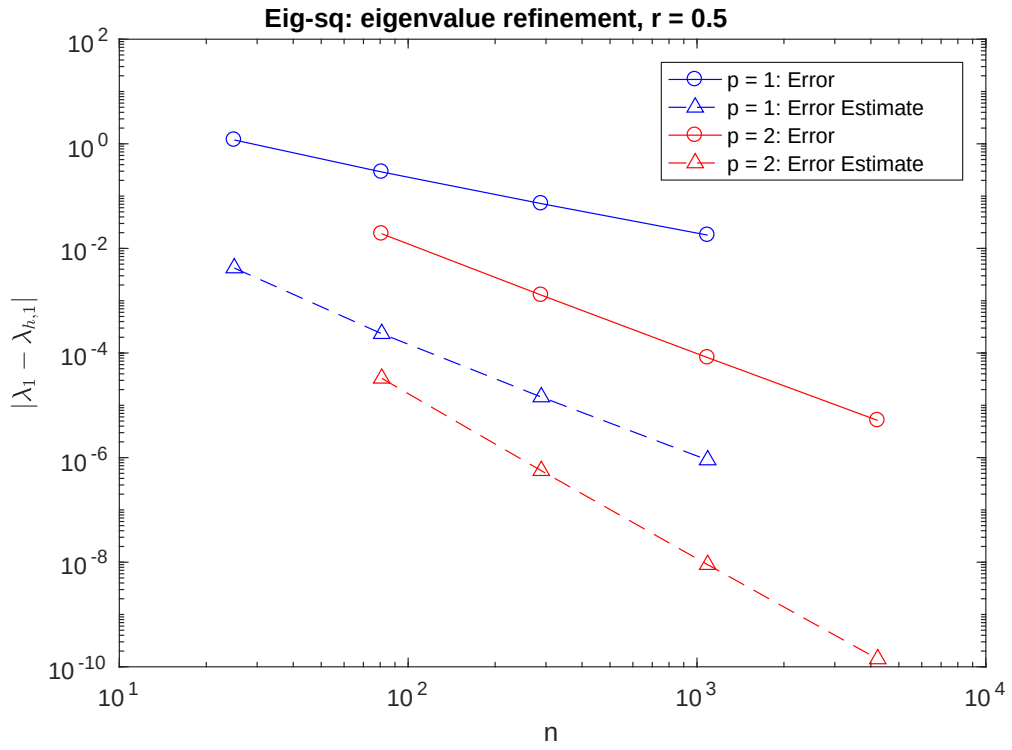


Figure 11: temp caption

Convergence rates for the above lines are shown in Table 5 below.

	$\mathbb{P}^1$	$\mathbb{P}^2$
Expected	-1.00	-2.00
Actual	-1.0492	-2.0670

Table 5: Convergence rates for eigenvalue error on square domain

Here it is clear that there is some scaling factor of error causing the large disagreement between the error and the error estimates. I'm not entirely sure what this is but the convergence rate is as expected for a functional output and the effectivities are quite constant.

(h) The reference eigenvalue was found to be $\lambda_1^{\text{ref}} = 9.639816656535080$ with an error estimate of $1.702667539918248 \times 10^{-4}$, this was computed using 8 iterations of our adaptive algorithm. The resulting first eigenfunction u_1 is shown in Figure 12

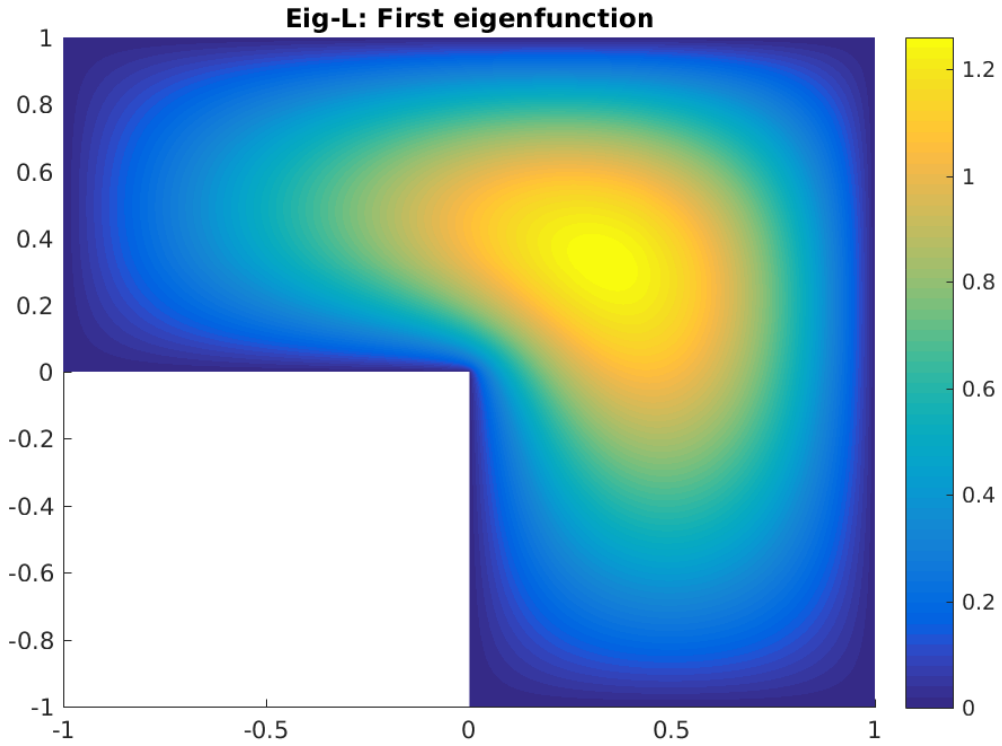


Figure 12: First eigenfunction on L-shaped membrane

(h) A convergence plot for the absolute value of the error in the first eigenvalue on the L-domain is shown in Figure 13.

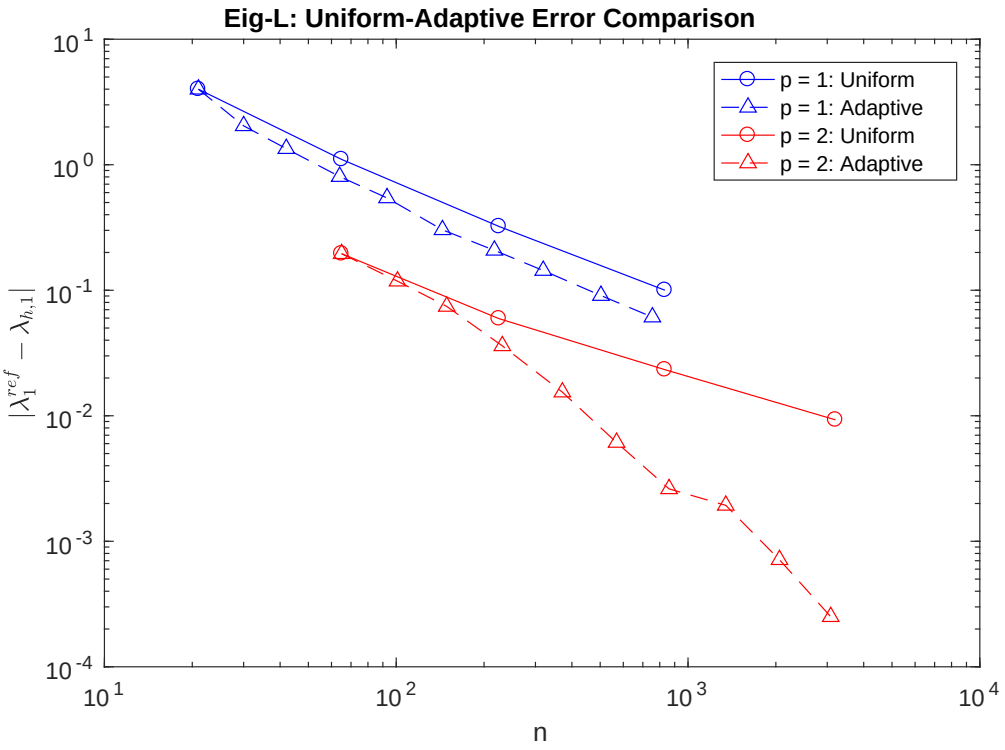


Figure 13: Square domain first eigenfunction L^2 norm convergence

Comparing uniform and adaptive refinement, we see evidence of a strong singularity in our domain. Where the uniform scheme cannot single out this region for more attention, the adaptive scheme does, and it allows us to achieve a higher convergence. These convergence rates are shown in Table 6 below. It further demonstrates that the convergence is highly limited by the regularity of the solution.

	$\mathbb{P}^1$	$\mathbb{P}^2$
Uniform	-0.8901	-0.6782
Adaptive	-1.3375	-2.1293

Table 6: Convergence rates for eigenvalue error on L domain

A convergence plot (h)(iii) is shown in Figure 14.

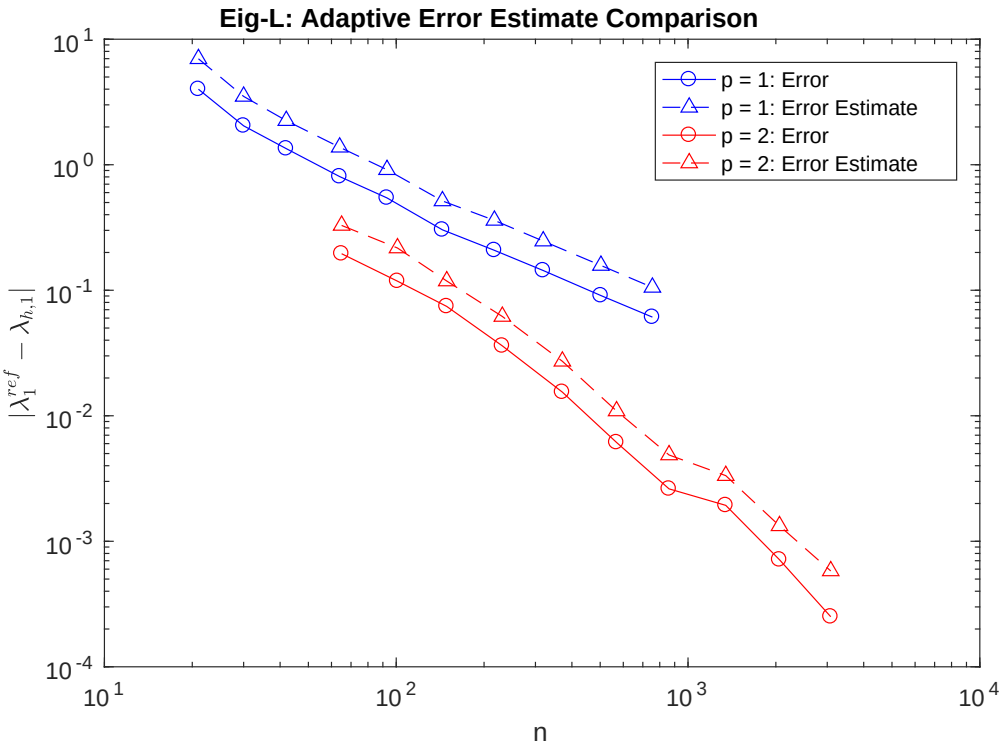


Figure 14: Square domain first eigenfunction L^2 norm convergence

The observed effectivity is comparable to that found for the square problem.

Lastly, we consider the mesh for our L-shaped eigenfunction shown in Figure 15, it shows a strong singularity at the corner located at the origin. This explains our poorer convergence rates as compared with the square domain, and we also note that a large portion of the refinement has been located here.

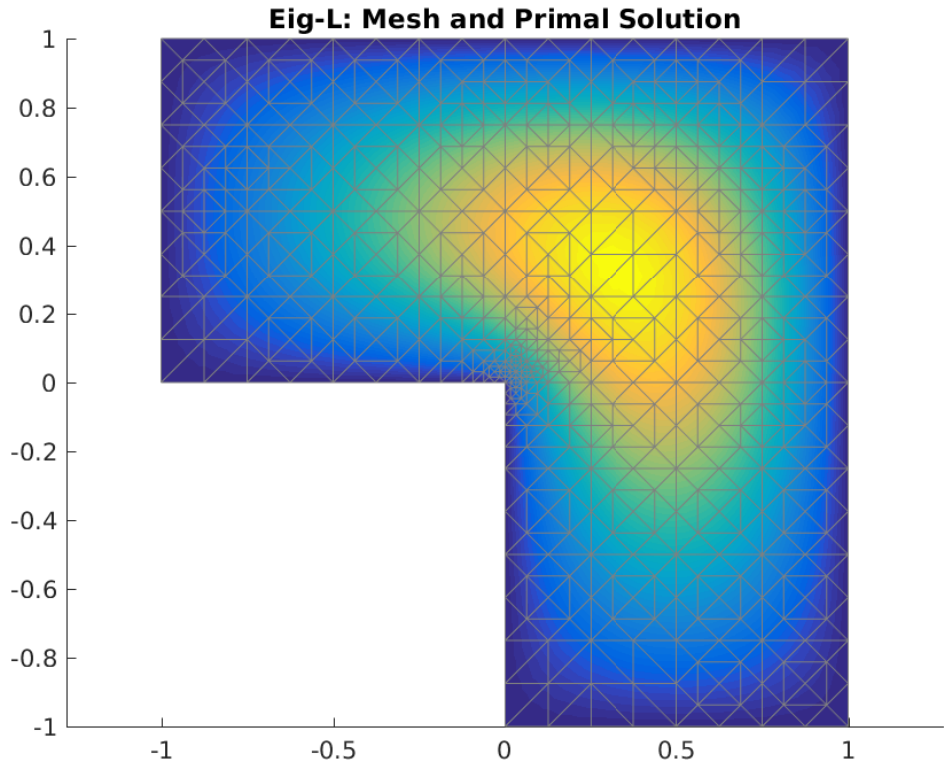


Figure 15: Square domain first eigenfunction L^2 norm convergence

(i) This is the MathWorks logo.

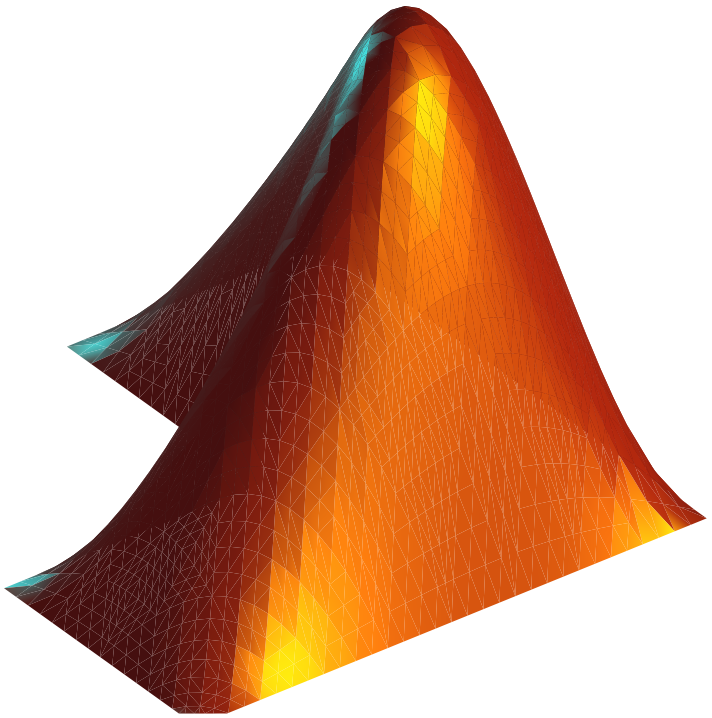


Figure 16: First eigenfunction on L-shaped membrane

Appendix

```
function ad_adapt_2d_template
% AD_ADAPT_2D_SIMPLE
% Copyright 2019 Masayuki Yano, University of Toronto

% discretization parameters
clr
tic
p = 2; % polynomial order
pquad = 2*p; % quadrature rule

% adaptation parameters
err_tol = 1e-9; % target V-norm relative error tolerance
niter = 4; % maximum number of iterations
reffrac = 0.1; % refinement fraction

% load equation
[eqn, mesh] = load_eqn_mesh('addiff-test',1/50);
% [eqn, mesh] = load_eqn_mesh('addiff', 1/50);
% [eqn, mesh] = load_eqn_mesh('eig-sq', 1);
% [eqn, mesh] = load_eqn_mesh('eig-L', 1);

% generate reference element
ref = make_ref_tri(p,pquad);

% update the mesh
if p == 2
    mesh = add_quadratic_nodes(mesh);
end
mesh = make_bgrp(mesh);

% Problem 1
if 0
% gls_flag = false;
% U = ad_solve(eqn,mesh,ref,gls_flag);
% figure(1)
% plot_field(mesh,ref,U);
% colorbar
% title('Galerkin solution to addiff-test equation')
% gls_flag = true;
% U_gls = ad_solve(eqn,mesh,ref,gls_flag);
% figure(2)
% plot_field(mesh,ref,U_gls);
% colorbar
% title('GLS solution to addiff-test equation')

L2_error = zeros(niter,2,2);
max_over = zeros(niter,2,2);
ndofs = zeros(niter,2,2);
for gls_flag = [false, true]
    for p = [1, 2]
        [eqn, mesh] = load_eqn_mesh('addiff-test',1/50);
        ref = make_ref_tri(p,pquad);
```

```
        if p == 2
            mesh = add_quadratic_nodes(mesh);
        end
        mesh = make_bgrp(mesh);

        for iter = 1:niter
            U = ad_solve(eqn,mesh,ref,gls_flag);
            ndofs(iter, p, gls_flag + 1) = length(U);
            [~,~,elemental_L2_error] = ad_soln_norm(eqn,mesh,ref,U);
            L2_error(iter,p,gls_flag + 1) = sqrt(sum(elemental_L2_error));
            max_over(iter,p,gls_flag + 1) = max(U-1);
            if iter ~= niter
                mesh = refine_uniform(mesh);
            end
        end
    end
end

figure(1)
loglog(ndofs(:,1,1),L2_error(:,1,1),'-^b',ndofs(:,2,1),L2_error(:,2,1),'-^
   r',ndofs(:,1,2),L2_error(:,1,2),'--ob',ndofs(:,2,2),L2_error(:,2,2),'--
   or')
title('Addiff-test L2 solution error convergence')
legend('p = 1, Galerkin','p = 2, Galerkin','p = 1, GLS','p = 2, GLS')
xlabel('n')
ylabel('$\|u - u_h\|_{L^2(\Omega)}$','interpreter','latex')

figure(2)
semilogx(ndofs(:,1,1),max_over(:,1,1),'-^b',ndofs(:,2,1),max_over(:,2,1),'
   -^r',ndofs(:,1,2),max_over(:,1,2),'--ob',ndofs(:,2,2),max_over(:,2,2),'
   --or')
title('Addiff-test L2 solution error convergence')
legend('p = 1, Galerkin','p = 2, Galerkin','p = 1, GLS','p = 2, GLS')
xlabel('n')
ylabel('$\max(\hat{u}_j-1)$','interpreter','latex')

conv_p1_SG = log(L2_error(end,1,1)./L2_error(end-2,1,1)) ./ log(ndofs(end
   ,1,1)/ndofs(end-2,1,1))
conv_p2_SG = log(L2_error(end,2,1)./L2_error(end-2,2,1)) ./ log(ndofs(end
   ,2,1)/ndofs(end-2,2,1))
conv_p1_GLS = log(L2_error(end,1,2)./L2_error(end-2,1,2)) ./ log(ndofs(end
   ,1,2)/ndofs(end-2,1,2))
conv_p2_GLS = log(L2_error(end-1,2,2)./L2_error(end-2,2,2)) ./ log(ndofs(
   end-1,2,2)/ndofs(end-2,2,2))
end

% Problem 2
if 0
[eqn, mesh] = load_eqn_mesh('addiff-test',1/50);

% generate reference element
ref = make_ref_tri(p,pquad);

% update the mesh
```

```
if p == 2
    mesh = add_quadratic_nodes(mesh);
end
mesh = make_bgrp(mesh);

[L2_errest_uniform, L2_err_uniform, L2_dofs_uniform] = ad_refine(eqn,mesh,
   ref,false,false,false,1,niter);
[L2_errest_adapt, L2_err_adapt, L2_dofs_adapt] = ad_refine(eqn,mesh,ref,
   false,false,false,reffrac,2*niter+1);

[eqn, mesh] = load_eqn_mesh('addiff', 1/50);
% generate reference element
ref = make_ref_tri(p,pquad);

% update the mesh
if p == 2
    mesh = add_quadratic_nodes(mesh);
end
mesh = make_bgrp(mesh);
L2_err_uniform = sqrt(L2_err_uniform);
L2_err_adapt= sqrt(L2_err_adapt);

[output_errest_uniform, output_uniform,output_dofs_uniform] = ad_refine(
   eqn,mesh,ref,false,true,true,1,niter);
[output_errest_L2, output_L2,output_dofs_L2] = ad_refine(eqn,mesh,ref,
   false,false,false,reffrac,2*niter+1);
[output_errest_adapt, output_adapt, output_dofs_adapt] = ad_refine(eqn,
   mesh,ref,false,true,true,reffrac,2*niter+1);

figure(3);
loglog(L2_dofs_uniform,L2_errest_uniform,'--ob',L2_dofs_adapt,
   L2_errest_adapt,'--^b',L2_dofs_uniform,L2_err_uniform,'-or',
   L2_dofs_adapt,L2_err_adapt,'-^r')
legend('Uniform: Estimate','Adapt: Estimate','Uniform: Error','Adapt:
   Error')

figure(4);
loglog(output_dofs_uniform,output_errest_uniform,'--ob',output_dofs_adapt,
   output_errest_adapt,'--^b',output_dofs_uniform,abs(output_uniform-eqn.
   sref),'-or',output_dofs_adapt,abs(output_adapt-eqn.sref),'-^r' )
legend('Uniform: Estimate','Adapt: Estimate','Uniform: Error','Adapt:
   Error')

figure(5);
loglog(output_dofs_uniform,output_errest_uniform,'--or',output_dofs_adapt,
   output_errest_adapt,'--^b',output_dofs_L2,output_errest_L2,'--sk',
   output_dofs_uniform,abs(output_uniform-eqn.sref),'-or',
   output_dofs_adapt,abs(output_adapt-eqn.sref),'-^b',output_dofs_L2,abs(
   output_L2-eqn.sref),'-sk' )
legend('Uniform: Estimate','Adapt: Estimate','L2: Estimate','Uniform:
   Error','Adapt: Error','L2: Error')

conv_L2_uniform = log(L2_errest_uniform(end)./L2_errest_uniform(end-1)) ./
    log(L2_dofs_uniform(end)/L2_dofs_uniform(end-1))
```

```
conv_L2_adapt = log(L2_errest_adapt(end)./L2_errest_adapt(end-2)) ./ log(
    L2_dofs_adapt(end)/L2_dofs_adapt(end-2))
conv_output_uniform = log(output_errest_uniform(end)./
    output_errest_uniform(end-1)) ./ log(output_dofs_uniform(end)/
    output_dofs_uniform(end-1))
conv_output_L2 = log(output_errest_L2(end)./output_errest_L2(end-1)) ./
    log(output_dofs_L2(end)/output_dofs_L2(end-1))
conv_output_adapt = log(output_errest_adapt(end)./output_errest_adapt(end
    -2)) ./ log(output_dofs_adapt(end)/output_dofs_adapt(end-2))
end

% Problem 3(1)
if 0

L2_error_est = zeros(niter,2);
L2_error = zeros(niter,2);
lambda_error_est = zeros(niter,2);
lambda_error = zeros(niter,2);
ndofs = zeros(niter,2);
for p = [1 2]
[eqn, mesh] = load_eqn_mesh('eig-sq', 1);

% generate reference element
ref = make_ref_tri(p,pquad);

% update the mesh
if p == 2
    mesh = add_quadratic_nodes(mesh);
end
mesh = make_bgrp(mesh);

[L2_error_est(:,p), lambda_error_est(:,p), ndofs(:,p), L2_error(:,p),
    lambda_error(:,p)] = ad_refine(eqn,mesh,ref,false,false,false,1,niter,
    true);
lambda_error_est = (lambda_error_est).^(1);
end

figure(6)
loglog(ndofs(:,1),L2_error(:,1),'-ob',ndofs(:,1),L2_error_est(:,1),'--^b',
    ndofs(:,2),L2_error(:,2),'-or',ndofs(:,2),L2_error_est(:,2),'--^r')
legend('p = 1: Error','p = 1: Error Estimate','p = 2: Error','p = 2: Error
    Estimate')

figure(7)
loglog(ndofs(:,1),lambda_error(:,1),'-ob',ndofs(:,1),lambda_error_est(:,1)
    ,'--^b',ndofs(:,2),lambda_error(:,2),'-or',ndofs(:,2),lambda_error_est
    (:,2),'--^r')
legend('p = 1: Error','p = 1: Error Estimate','p = 2: Error','p = 2: Error
    Estimate')

conv_L2_eigfun = log(L2_error_est(end,:)./L2_error_est(end-1,:)) ./ log(
    ndofs(end,:)/ndofs(end-1,:))
conv_lambda_eigfun = log(lambda_error_est(end,:)./lambda_error_est(end
    -1,:)) ./ log(ndofs(end,:)/ndofs(end-1,:))
```

```
end

% Problem 3(2)
if 1

L2_error_est = zeros(niter,2);
L2_error = zeros(niter,2);
lambda_error_est = zeros(niter,2);
lambda_error = zeros(niter,2);
ndofs = zeros(niter,2);
for p = [1 2]
[eqn, mesh] = load_eqn_mesh('eig-L', 1);

% generate reference element
ref = make_ref_tri(p,pquad);

% update the mesh
if p == 2
    mesh = add_quadratic_nodes(mesh);
end
mesh = make_bgrp(mesh);

[L2_error_est(:,p), lambda_error_est(:,p), ndofs(:,p), L2_error(:,p),
   lambda_error(:,p)] = ad_refine(eqn,mesh,ref,false,false,false,1,niter,
   true);
end



% conv_L2_eigfun = log(L2_error_est(end,:)./L2_error_est(end-1,:)) ./ log(
   ndofs(end,:)/ndofs(end-1,:))
conv_lambda_eigfun = log(lambda_error_est(end,:)./lambda_error_est(end
   -1,:)) ./ log(ndofs(end,:)/ndofs(end-1,:))

niter = floor(1.5*niter);
L2_error_est_adapt = zeros(niter,2);
L2_error_adapt = zeros(niter,2);
lambda_error_est_adapt = zeros(niter,2);
lambda_error_adapt = zeros(niter,2);
ndofs_adapt = zeros(niter,2);
for p = [1 2]
[eqn, mesh] = load_eqn_mesh('eig-L', 1);

% generate reference element
ref = make_ref_tri(p,pquad);

% update the mesh
if p == 2
    mesh = add_quadratic_nodes(mesh);
end
mesh = make_bgrp(mesh);

[L2_error_est_adapt(:,p), lambda_error_est_adapt(:,p), ndofs_adapt(:,p),
```

```
    L2_error_adapt(:,p), lambda_error_adapt(:,p)] = ad_refine(eqn,mesh,ref,
    false,false,false,0.1,niter,true);
% lambda_error = (lambda_error).^2;
end

figure(6)
loglog(ndofs(:,1),lambda_error(:,1),'-ob',ndofs_adapt(:,1),
    lambda_error_adapt(:,1),'--^b',ndofs(:,2),lambda_error(:,2),'-or',
    ndofs_adapt(:,2),lambda_error_adapt(:,2),'--^r')
legend('p = 1: Uniform','p = 1: Adaptive','p = 2: Uniform','p = 2:
    Adaptive')
xlabel('n')
ylabel('$|\lambda^{ref}_1 - \lambda_{h,1}|$','interpreter','latex')
title('Eig-L: Uniform-Adaptive Error Comparison')

figure(7)
loglog(ndofs_adapt(:,1),lambda_error_adapt(:,1),'-ob',ndofs_adapt(:,1),
    lambda_error_est_adapt(:,1),'--^b',ndofs_adapt(:,2),lambda_error_adapt
    (:,2),'-or',ndofs_adapt(:,2),lambda_error_est_adapt(:,2),'--^r')
legend('p = 1: Error','p = 1: Error Estimate','p = 2: Error','p = 2: Error
     Estimate')
xlabel('n')
ylabel('$|\lambda^{ref}_1 - \lambda_{h,1}|$','interpreter','latex')
title('Eig-L: Adaptive Error Estimate Comparison')

% conv_L2_eigfun = log(L2_error_est_adapt(end,:)./L2_error_est_adapt(end
    -1,:)) ./ log(ndofs_adapt(end,:)/ndofs_adapt(end-1,:))
conv_lambda_eigfun = log(lambda_error_est_adapt(end,:)./
    lambda_error_est_adapt(end-1,:)) ./ log(ndofs_adapt(end,:)/ndofs_adapt(
    end-1,:))

end

toc
end

function [error_estimate, outputs, ndofs, eig_error, lambda_error] =
    ad_refine(eqn,mesh0,ref,gls_flag,output_flag,H1_flag,reffrac,niter,
    eigen_flag)
% AD_REFINE solves the finite element problem on succesively refined
    meshes
% INPUT:
%   eqn: equation structure; the exact solution is grabbed from here
%   mesh: mesh structure
%   ref: reference structure
%   gls_flag: boolean flag that activates GLS stabilization
%   output_flag: true if controling output error, false if L2 soln error
%   H1_flag: true if using H1 norm of extrap error, false if L2
%   reffrac: refinement fraction, 1 if uniform refinement
% OUTPUT:
%   errors: error estimates at each mesh
%   ndofs: number of degrees of freedom associated with each error est
if nargin < 9
    eigen_flag = false;
```

```
end

error_estimate = zeros(niter,1);
outputs = zeros(niter,1);
ndofs = zeros(niter,1);
if eigen_flag
   eig_error = zeros(niter,1);
   lambda_error = zeros(niter,1);
end
for iter = 1:niter
    % SOLVE
    if ~eigen_flag
        [U0, ~, Psi0] = ad_solve(eqn,mesh0,ref,gls_flag);
    else
        [U0, lambda0] = ad_eigen_solve(eqn,mesh0,ref,1);
    end

    % ESTIMATE
    [mesh1,Ue] = refine_uniform(mesh0,U0);

    if ~eigen_flag
        [~,Psie] = refine_uniform(mesh0,Psi0);
        [U1, outputs(iter), Psi1] = ad_solve(eqn,mesh1,ref,gls_flag);
    else
        [U1, lambda1] = ad_eigen_solve(eqn,mesh1,ref,1);
        outputs(iter) = abs(lambda1 - lambda0)/(2^0.5-1);
    end
    if H1_flag && output_flag
        elem_err_est_1 = (ad_soln_norm(eqn, mesh1, ref, Psie-Psi1)).*(
           ad_soln_norm(eqn, mesh1, ref, Ue-U1));
    elseif H1_flag && ~output_flag
        elem_err_est_1 = ad_soln_norm(eqn, mesh1, ref, Ue - U1);
    else
        [~,elem_err_est_1] = ad_soln_norm(eqn, mesh1, ref, Ue - U1);
        if isfield(eqn,'ufun')
            [~,~,l1l2_err_true] = ad_soln_norm(eqn,mesh0,ref,U0);
            outputs(iter) = sum(l1l2_err_true);
        end
    end
    ntri = size(mesh0.tri,1);
    elem_err_est_0 = zeros(ntri,1);
    for elem = 1:ntri
        idx = [elem elem+ntri elem+2*ntri elem+3*ntri];
        elem_err_est_0(elem) = sum(elem_err_est_1(idx));
    end

    if eqn.bvalue(1) == 1 % addiff-test
        error_estimate(iter) = sqrt(sum(elem_err_est_0))/(2^0.5-1);
    else
        error_estimate(iter) = sqrt(sum(elem_err_est_0))/(2^1-1);
    end
    ndofs(iter) = length(U0);

    if eigen_flag
```

```
        lambda_error(iter) = abs(lambda0-eqn.sref);
        [~,~,temp] = ad_soln_norm(eqn,mesh0,ref,U0);
        eig_error(iter) = sqrt(sum(temp));
    end

    % MARK & REFINE
    if reffrac == 1
        mesh0 = refine_uniform(mesh0);
    else
        err_est_sorted = sort(elem_err_est_0);
        tmark = zeros(ntri,1);
        tmark(elem_err_est_0 > err_est_sorted(end-floor(reffrac*ntri))) =
           1;
        mesh0 = refine_mesh_adapt(mesh0,tmark);
    end

    if iter == niter
       figure(1)
       plot_field(mesh1,ref,U1,struct('edgecolor',[0.5,0.5,0.5]));
       my_plot_lshaped(mesh1,ref,U1)
    end

end
end

function [A,M,F,Fo] = ad_assemble(eqn,mesh,ref,gls_flag)
% AD_ASSEMBLE assembles the stiffness matrix, load vector, and norm matrix
% INPUT:
%   eqn: equation structure; the exact solution is grabbed from here
%   mesh: mesh structure
%   ref: reference structure
%   gls_flag: boolean flag that activates GLS stabilization
% OUTPUT:
%   A: stiffness matrix
%   M: mass matrix
%   F: load vector
%   Fo: output vector
% NOTE:
%   None of the output operators incoporate the essential boundary
%   conditions.

% get useful parameters
dim = 2;
[nelem,nshp] = size(mesh.tri);
nq = length(ref.wq);

% compute and store local matrices
amat = zeros(nshp,nshp,nelem);
mmat = zeros(nshp,nshp,nelem);
imat = zeros(nshp,nshp,nelem);
jmat = zeros(nshp,nshp,nelem);
fvec = zeros(nshp,nelem);
fovec = zeros(nshp,nelem);
ivec = zeros(nshp,nelem);
```

```
for elem = 1:nelem
    tril = mesh.tri(elem,:).';

    % compute mesh jacobians
    xl = mesh.coord(tril,:);
    xq = ref.shp*xl;
    jacq = zeros(nq,dim,dim);
    for j = 1:dim
        jacq(:,:,j) = ref.shpx(:,:,j)*xl;
    end
    detJq = jacq(:,1,1).*jacq(:,2,2) - jacq(:,1,2).*jacq(:,2,1);
    ijacq = zeros(nq,dim,dim);
    ijacq(:,1,1) =  1./detJq.*jacq(:,2,2);
    ijacq(:,1,2) = -1./detJq.*jacq(:,1,2);
    ijacq(:,2,1) = -1./detJq.*jacq(:,2,1);
    ijacq(:,2,2) =  1./detJq.*jacq(:,1,1);

    % compute quadrature weight
    wqJ = ref.wq.*detJq;

    % compute basis
    phiq = ref.shp;
    phixq = zeros(nq,nshp,dim);
    for j = 1:dim
        for k = 1:dim
            phixq(:,:,j) = phixq(:,:,j) + bsxfun(@times,ref.shpx(:,:,k),
                ijacq(:,k,j));
        end
    end

    if gls_flag
        phixxq = zeros(nq,nshp,dim,dim);
        phixxq(:,:,1,1) = bsxfun(@times,ijacq(:,1,1),bsxfun(@times,ref.
            shpxx(:,:,1,1),ijacq(:,1,1))) + ...
             bsxfun(@times,ijacq(:,2,1),bsxfun(@times,ref.shpxx(:,:,2,1),
                ijacq(:,1,1))) + ...
             bsxfun(@times,ijacq(:,1,1),bsxfun(@times,ref.shpxx(:,:,2,1),
                ijacq(:,2,1))) + ...
             bsxfun(@times,ijacq(:,2,1),bsxfun(@times,ref.shpxx(:,:,2,2),
                ijacq(:,2,1)));

        phixxq(:,:,2,2) = bsxfun(@times,ijacq(:,1,2),bsxfun(@times,ref.
            shpxx(:,:,1,1),ijacq(:,1,2))) + ...
             bsxfun(@times,ijacq(:,2,2),bsxfun(@times,ref.shpxx(:,:,2,1),
                ijacq(:,1,2))) + ...
             bsxfun(@times,ijacq(:,1,2),bsxfun(@times,ref.shpxx(:,:,1,2),
                ijacq(:,2,2))) + ...
             bsxfun(@times,ijacq(:,2,2),bsxfun(@times,ref.shpxx(:,:,2,2),
                ijacq(:,2,2)));
    end

    % compute stiffness matrix
    aaloc = zeros(nshp,nshp);
    mmloc = zeros(nshp,nshp);
```

```
    ffloc = zeros(nshp,1);
    foloc = zeros(nshp,1);
    b = eqn.bfun(xq);
    kap = eqn.kfun(xq);
    L = zeros(nq,nshp);
    for i = 1:dim
        aaloc = aaloc + phixq(:,:,i)'*diag(eqn.kfun(xq).*wqJ)*phixq(:,:,i)
            ;
        aaloc = aaloc - phixq(:,:,i)'*diag(b(:,i).*wqJ)*phiq;
        mmloc = mmloc + phiq'*diag(wqJ)*phiq;
        if gls_flag
            h = max(sqrt((detJq)));
            tau = 1/sqrt((2*b(1)/h)^2 + 9*(4*kap(1)/h^2)^2);
            L = L + diag(b(:,i))* phixq(:,:,i);
            L = L - diag(kap)*phixxq(:,:,i,i);
            aaloc = aaloc + L'*diag(wqJ.*tau)*L;
            ffloc = ffloc + L'*(wqJ.*tau);
            foloc = foloc + L'*(wqJ.*tau);
        end
    end


    % compute load vector
    ffloc = ffloc + phiq'*(wqJ.*eqn.ffun(xq));

    % compute output load vector
    foloc = foloc + phiq'*(wqJ.*eqn.fofun(xq));

    % insert to global matrix
    amat(:,:,elem) = aaloc;
    mmat(:,:,elem) = mmloc;
    imat(:,:,elem) = repmat(tril,[1,nshp]);
    jmat(:,:,elem) = repmat(tril',[nshp,1]);
    fvec(:,elem) = ffloc;
    fovec(:,elem) = foloc;
    ivec(:,elem) = tril;
end

% boundary conditions
for bgrp = 1:length(mesh.bgrp)
    for edge = 1:size(mesh.bgrp{bgrp},1)
        elem = mesh.bgrp{bgrp}(edge,3);
        ledge = mesh.bgrp{bgrp}(edge,4);
        lnode = ref.f2n(:,ledge);
        tril = mesh.tri(elem,lnode).';

        % compute mesh jacobians
        xl = mesh.coord(tril,:);
        %xq = ref.shpf*xl;
        jacq = ref.shpxf*xl;
        detJq = sqrt(sum(jacq.^2,2));

        % compute quadrature weight
        wqJ = ref.wqf.*detJq;
```

```
        % compute basis
        phiq = ref.shpf;

        % root inhomogeneous Neumann condition
        % Could actually compute true normal vector here
        if (eqn.btype(bgrp) == 'n')
            ffloc = phiq'*(wqJ.*0.0);
            fvec(lnode,elem) = fvec(lnode,elem) + ffloc;
        end
    end
end

% assemble matrix
ndof = size(mesh.coord,1);
A = sparse(imat(:),jmat(:),amat(:),ndof,ndof);
M = sparse(imat(:),jmat(:),mmat(:),ndof,ndof);
F = accumarray(ivec(:),fvec(:));
Fo = accumarray(ivec(:),fovec(:));

end

function [U,s,Psi] = ad_solve(eqn,mesh,ref,gls_flag)
% AD_SOLVE solves the finite element problem
% INPUT:
%   eqn: equation structure; the exact solution is grabbed from here
%   mesh: mesh structure
%   ref: reference structure
%   gls_flag: boolean flag that activates GLS stabilization
% OUTPUT:
%   U: solution
%   s: output
%   Psi: adjoint solution

ndof = size(mesh.coord,1);
[A,~,F,Fo] = ad_assemble(eqn,mesh,ref,gls_flag);

U = zeros(ndof,1);
Psi = zeros(ndof,1);

% identify internal and boundary nodes
bnodes = [];
for bgrp = 1:length(mesh.bgrp)
    if (eqn.btype(bgrp) == 'd')
        bnodes = [bnodes; nodes_on_boundary(mesh,ref,bgrp)];
        U(nodes_on_boundary(mesh,ref,bgrp)) = eqn.bvalue(bgrp);
    end
end
% bnodes = bnodes(:);
inodes = setdiff((1:ndof)', bnodes);

% solve linear system
U(inodes) = A(inodes,inodes)\(F(inodes) - A(inodes,bnodes)*U(bnodes));
Psi(inodes) = A(inodes,inodes)'\(Fo(inodes) - A(inodes,bnodes)*U(bnodes));
```

```
% plot solution
if (0)
    figure(1), clf,
    plot_field(mesh,ref,U,struct('edgecolor',[0.5,0.5,0.5]));
    colorbar
end

s = Fo'*U;
% fprintf('s = %4.4e\n',s);

end

function [U,lambda] = ad_eigen_solve(eqn,mesh,ref,k)
% AD_EIGEN_SOLVE solves the finite element eigenproblem
% INPUT:
%   eqn: equation structure; the exact solution is grabbed from here
%   mesh: mesh structure
%   ref: reference structure
%   k: eigenpair index
% OUTPUT:
%   U: eigenfunction
%   lambda: eigenvalue
if nargin < 4
    k = 1;
end


[A,M,~,~] = ad_assemble(eqn,mesh,ref,false);

ndof = size(mesh.coord,1);
U = zeros(ndof,1);

% identify internal and boundary nodes
bnodes = [];
for bgrp = 1:length(mesh.bgrp)
    if (eqn.btype(bgrp) == 'd')
        bnodes = [bnodes; nodes_on_boundary(mesh,ref,bgrp)];
    end
end
% bnodes = bnodes(:);
inodes = setdiff((1:ndof)', bnodes);

Ah = (A(inodes,inodes)+A(inodes,inodes)');

[V,lambda] = eigs(Ah,-M(inodes,inodes),k,'SM');
normalizer = V' * M(inodes,inodes) * V;
U(inodes,:) = sqrt(2)*V/sqrt(normalizer);
U(bnodes,:) = 0;
lambda = -diag(lambda);
end

function [hh1,ll2,ll2_true,hh1_true] = ad_soln_norm(eqn, mesh, ref, U)
% AD_SOLN_NORM computes norm of the solution field
```

```
% INPUT:
%   eqn: equation structure; the exact solution is grabbed from here
%   mesh: mesh structure
%   ref: reference structure
%   U: solution coefficient
% OUTPUT:
%   hh1: square of the element-wise H^1 norm of U.  Array of length
%        size(mesh.tri,1).
%   ll2: square of the element-wise L^2 norm of U.
%   ll2_true: square of the element-wise L^2 norm of the error in U.
%             Requires eqn.ufun to be specified.
dim = 2;
[nelem,nshp] = size(mesh.tri);

hh1 = zeros(nelem,1);
ll2 = zeros(nelem,1);
ll2_true = zeros(nelem,1);
hh1_true = zeros(nelem,1);
for elem = 1:nelem
    % Define preliminaries
    tril = mesh.tri(elem,:).';
    xl = mesh.coord(tril,:);
    xq = ref.shp*xl;
    jacq = zeros(length(ref.wq),dim,dim);
    for j = 1:dim
        jacq(:,:,j) = ref.shpx(:,:,j)*xl;
    end
    detJq = jacq(:,1,1).*jacq(:,2,2) - jacq(:,1,2).*jacq(:,2,1);
    nq = length(ref.wq);
    ijacq = zeros(nq,dim,dim);
    ijacq(:,1,1) =  1./detJq.*jacq(:,2,2);
    ijacq(:,1,2) = -1./detJq.*jacq(:,1,2);
    ijacq(:,2,1) = -1./detJq.*jacq(:,2,1);
    ijacq(:,2,2) =  1./detJq.*jacq(:,1,1);
    phixq = zeros(nq,nshp,dim);
    for j = 1:dim
        for k = 1:dim
            phixq(:,:,j) = phixq(:,:,j) + bsxfun(@times,ref.shpx(:,:,k),
               ijacq(:,k,j));
        end
    end

    % compute quadrature weight
    wqJ = ref.wq.*detJq;

    % Evaluate true and approximate solutions on element
    U_ll2 = ref.shp*U(tril);
    U_hh1_1 = phixq(:,:,1)*U(tril);
    U_hh1_2 = phixq(:,:,2)*U(tril);

    % Add element wise contribution of error norm
    ll2(elem) = U_ll2'*(wqJ.*U_ll2);
    hh1(elem) = ll2(elem) + U_hh1_1'*(wqJ.*U_hh1_1) + U_hh1_2'*(wqJ.*
       U_hh1_2);
```

```
    if isfield(eqn,'ufun')
        ll2_true(elem) = (eqn.ufun(xq(:,1),xq(:,2)) - U_ll2)'*(wqJ.*(eqn.
            ufun(xq(:,1),xq(:,2)) - U_ll2));
        if isfield(eqn,'dufun')
            hh1_true(elem) = ll2_true(elem) + (eqn.dufun(xq) - U_hh1_1)'*(
                wqJ.*(eqn.dufun(xq) - U_hh1_1)) + (0 - U_hh1_2)'*(wqJ.*(0 -
                 U_hh1_2));
        end
    end
end
end

function [eqn, mesh] = load_eqn_mesh(eqnname, param)
% LOAD_EQN_MESH loads the equation and mesh structure
% INPUT
%   eqnname: string associated with problem type
%   param: optional parameters for the specific problem
% OUTPUT
%   eqn: equation structure
%      .k: diffusion function
%      .b: advection function
%      .f: source function
%      .fo: output weight function
%      .bytpe: bytype(i) \in {'d','n'}, where 'd' is Dirichlet and 'n' is
%              Neumann.
%      .bvalue: boundary values for Dirichlet or Neumann condition
%      .ufun: exact solution/eigenfunction
%      .sref: reference output
% NOTE:
%   problems
%   - addiff-test: advection-diffusion equation with known solution
%                  (param = diffusion coeff)
%   - addiff: advection-diffusion equation (param = diffusion coeff)
%   - eig-sq: eigenproblem on a square domain (param = eigenmode)
%   - eig-L: L-shaped domain eigenproblem (param = eigenmode)
%
switch lower(eqnname)
    case 'addiff-test'
        kap = param(1);
        eqn.kfun = @(x) kap*ones(size(x,1),1);
        eqn.bfun = @(x) [ones(size(x,1),1), zeros(size(x,1),1)];
        eqn.ffun = @(x) zeros(size(x,1),1);
        eqn.fofun = @(x) ones(size(x,1),1);
        eqn.btype = ['d','d','n','n'];
        eqn.bvalue = [1,0,0,0];

        eqn.ufun = @(x1,x2) (exp(x1/kap) - exp(1/kap))/(1-exp(1/kap));
        eqn.dufun = @(x) (exp(x(:,1)/kap)/kap)/(1-exp(1/kap));

        mesh = make_square_mesh(1/4,'unstructured');
    case 'addiff'
        kap = param(1);
        eqn.kfun = @(x) kap*ones(size(x,1),1);
        eqn.bfun = @(x) [ones(size(x,1),1), ones(size(x,1),1)];
```

```
        eqn.ffun = @(x) ones(size(x,1),1);
        eqn.fofun = @(x) ones(size(x,1),1);
        eqn.btype = ['d','d','d','d'];
        eqn.bvalue = [0,0,0,0];
        if (kap == 1/50)
            eqn.sref = 2.984344669078263e-01;
        end

        mesh = make_square_mesh(1/4,'unstructured');
    case 'eig-sq'
        eqn.kfun = @(x) ones(size(x,1),1);
        eqn.bfun = @(x) zeros(size(x,1),2);
        eqn.ffun = @(x) zeros(size(x,1),1);
        eqn.fofun = @(x) zeros(size(x,1),1);
        eqn.btype = ['d','d','d','d'];
        eqn.bvalue = [0,0,0,0];

        eqn.ufun = @(x1,x2) 2*sin(pi*x1).*sin(pi*x2);
        switch param(1)
            case 1
                eqn.sref = 2*pi^2;
        end

        mesh = make_square_mesh(1/4,'unstructured');
    case 'eig-l'
        eqn.kfun = @(x) ones(size(x,1),1);
        eqn.bfun = @(x) zeros(size(x,1),2);
        eqn.ffun = @(x) zeros(size(x,1),1);
        eqn.fofun = @(x) zeros(size(x,1),1);
        eqn.btype = ['d','d','d'];
        eqn.bvalue = [0,0,0];
        switch param(1)
            case 1
                eqn.sref = 9.639816656535080;
        end
        mesh = make_lshaped_mesh;
        mesh = refine_mesh_adapt(mesh, ones(size(mesh.tri,1),1));
    otherwise
        error('unsupported problem type');
end
end

function mesh = make_lshaped_mesh
% MAKE_LSHAPED_MESH creates a mesh for a L-shaped domain
% OUTPUT
%   mesh: mesth structure
% REMARKS
%   boundary groups
%     1: innner boundary
%     2: outer boundary
%     3: side
%
coord = [0,0
         0,-1
```

```
          1,-1
          1,0
          1,1
          0,1
          -1,1
          -1,0];
tri = [1,2,3
       1,3,4
       1,4,5
       1,5,6
       1,6,7
       1,7,8];
bgrp{1} = [8,1
           1,2];
bgrp{2} = [3,4
           4,5
           5,6
           6,7];
bgrp{3} = [2,3
           7,8];
mesh.coord = coord;
mesh.tri = tri;
mesh.bgrp = bgrp;

end

function my_plot_lshaped(mesh,ref,U)
% Pretty plot for the L-shaped domain problem.
% Copyright 1984-2007 The MathWorks, Inc.
%
figh = figure(101);
clf;
opt = struct('surf',true);
mesh.coord(:,2) = -mesh.coord(:,2);
mesh.coord = 0.5*bsxfun(@plus,mesh.coord,[1,1]);
mesh.coord = 50*mesh.coord;
U = 40*U/max(abs(U))*sign(sum(U));
h = plot_field(mesh,ref,U,opt); %axis equal;

if (false)
    mesh2 = make_square_mesh(1);
    mesh2.coord = 50*(mesh2.coord);
    if p == ref.p
        mesh2 = add_quadratic_nodes(mesh2);
    end
    h2 = plot_field(mesh2,ref,zeros(size(mesh2.coord,1),1),opt);
    h = [h; h2];
end

axis off;

logoax = axes('CameraPosition', [-193.4013 -265.1546  220.4819],...
    'CameraTarget',[26 26 10], ...
    'CameraUpVector',[0 0 1], ...
```

```
    'CameraViewAngle',9.5, ...
    'DataAspectRatio', [1 1 .9],...
    'Position',[0 0 1 1], ...
    'Visible','off', ...
    'XLim',[1 51], ...
    'YLim',[1 51], ...
    'ZLim',[-13 40], ...
    'parent',figh);
s = set(h, ...
    'EdgeColor','none', ...
    'FaceColor',[0.9 0.2 0.2], ...
    'FaceLighting','phong', ...
    'AmbientStrength',0.3, ...
    'DiffuseStrength',0.6, ...
    'Clipping','off',...
    'BackFaceLighting','lit', ...
    'SpecularStrength',1, ...
    'SpecularColorReflectance',1, ...
    'SpecularExponent',7, ...
    'parent',logoax);
l1 = light('Position',[40 100 20], ...
    'Style','local', ...
    'Color',[0 0.8 0.8], ...
    'parent',logoax);
l2 = light('Position',[.5 -1 .4], ...
    'Color',[0.8 0.8 0], ...
    'parent',logoax);
%z = zoom(figh);
%z.setAxes3DPanAndZoomStyle(logoax,'camera');
end
```